Evaluating Binary Polynomials using Subpolynomials

Jacob M. Howe[1], Martin Brain[1] and Arnau Gàmez-Montolio[1]

[1]City St George's, University of London, UK

Abstract

Polynomials over bit-vectors, binary polynomials, are considered algebraically and it is observed that binary polynomials can be considered as a tree of subpolynomials. A construction of such subpolynomials is given. The application of these subpolynomials, along with other recent results on binary polynomials, in encoding polynomials over bit-vectors as part of solving SMT problems is investigated. Preliminary results are given.

Keywords

Polynomials, Satisfiability Modulo Theories, theory of bit-vectors, bit-blasting, modulo arithemetic, finite rings

1. Introduction

Evaluation of polynomials is a fundamental computational process. Polynomials are evaluated over some numerical or algebraic structure, be that fields or integer rings. Numbers need to be represented on a machine and this representation is usually in terms of bit-vectors. Polynomials over bit-vectors are referred to as *binary polynomials*. This paper is concerned with the evaluation of binary polynomials, with a particular interest in how this can be exploited as a component of solving SMT problems.

First-order terms and predicates in the theory of bit-vectors are typically handled in SMT solvers through *bit-blasting*, reducing numbers represented as bit-vectors to their constituent bits and reasoning propositionally about these. This means that reasoning about bit-vectors can take advantage of powerful SAT solving techniques. However, it is not clear that this is the right way to handle expressions involving multiplication since this can lead to some rather large representations (and/or representations that do not perform well). A short survey of the state-of-the-art, along with some suggestions as to improvements and a critique of the whole approach, can be found in [1].

Evaluating a polynomial potentially involves a lot of multiplication, so there is some hope that investigating alternative ways of handling polynomials might lead to a big pay-off in terms of the performance of SMT problems over bit-vectors. This might be particularly important since non-linear problems represent a hard challenge within formal verification, with SMT solvers being the main engine for doing this.

The theory of bit-vectors contains several sub-algebras. If bit-vectors are interpreted as unsigned binary numbers then there is a ring isomorphism with integers modulo 2^w (where w is the bit-width). This means that binary polynomials also form a ring. The challenge is to leverage the algebraic structure of binary polynomials to better manipulate inputs so that when they are eventually bit-blasted evaluation within SMT solvers is improved.

Noting that bit-vectors modulo 2^w are equivalent to finite rings, two main results are observed and exploited in this work. Firstly, the observations made in [2] are reiterated, in particular that binary polynomials can be expressed in a normal form with a maximum degree dependent on the bit-width. Secondly, a new technique is introduced noting the finite nature of binary polynomials means that a polynomial can be broken down into smaller *subpolynomials*; the construction and algorithms for doing so are presented.

The complexity of polynomials pulls in several directions. Firstly, there is the degree of the polynomial being considered, which initially appears to be potentially unbounded. Secondly, there is the number of

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK

j.m.howe@city.ac.uk (J. M. Howe); martin.brain@city.ac.uk (M. Brain); arnau.gamez-montolio@city.ac.uk (A. Gàmez-Montolio)

0000-0001-8013-6941 (J. M. Howe); 0000-0003-4216-7151 (M. Brain); 0009-0003-7676-4218 (A. Gàmez-Montolio)

variables that the polynomial is over. Thirdly, there is the number and complexity of the monomials that are summed to give the polynomials (clearly this is partly dependent on the number of variables that the polynomial is over). Fourthly, there is the bit-width of the bit-vectors. This work restricts itself to *univariate* polynomials (that is, polynomials over one variable), hence monomials are simply powers of the variable; some pointers to results for multivariate polynomials will be given. Results concerning the degree of the polynomials considered are given in Section 2. Ultimately, interest is in typical machine representations of unsigned integers (8, 16, 32, 64 and 128 bits) and floating-point (24/48 and 53/106 bits), though worked examples will illustrate the arguments over smaller bit-widths.

The first part revisits the results from [2] and contextualises with other related literature. The second part investigates how polynomials change when their bit-width is reduced. The third part considers how the observations made in the first and second parts can be exploited in the encoding of polynomials for bit-blasting in SMT.

This paper makes the following contributions:

- Introduces subpolynomials and investigates how they may be used to evaluate polynomials.
- Applies recent results from [2] to reduce binary polynomials to a normal form with a maximum degree.
- Performs an initial empirical investigation into the application of the above when solving SMT problems involving polynomials over bit-vectors.

The rest of the paper is structured as follows. Section 2 gives background and notation as well as surveying some of the related literature, in particular recent work on finding the maximum order for binary polynomials, given the bit-width. Section 3 gives results on polynomials for varying bit-width, Section 4 investigates how this can be exploited for evaluating polynomials over bit-vectors, Section 5 gives initial empirical results on this and Section 6 concludes.

2. Background and Related Work

This section formalises notation, whilst also surveying related work in the area. In particular, firstly results from [2] are summarised and illustrated, and secondly an overview of the evaluation of polynomials in general and in the context of SMT is provided.

2.1. Notation

A *bit-vector* with *bit-width* w consists of a vector of 0/1 values denoted $b = [b_{w-1}, b_{w-2}, ..., b_2, b_1, b_0]$. This can be interpreted as an unsigned integer $b = 2^{w-1}b_{w-1} + 2^{w-2}b_{w-2} + ... + 2^2b_2 + 2b_1 + b_0$. These are treated as integers modulo 2^w, that is, $-b = 2^w - b$. Being integer rings, there is no division operation, however, in some places something akin to division will be required in the form of a (zero-fill or logical) bit shift to the right, $\mathsf{shR}(b) = [0, b_{w-1}, b_{w-2}, ..., b_2, b_1]$.

A monomial is a product of variables, possibly raised to some power, and a coefficient. Powers are natural numbers and coefficients are integers. A polynomial is then a sum of monomials. This work considers only univariate polynomials (that is, with only one variable, hence monomials are a power of that variable multiplied by a coefficient). Hence polynomials have the form:

$$a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

where $n \in \mathbb{N}$ and $a_i \in \mathbb{Z}$ (in much of this work coefficients will be reduced modulo 2^w). Such a polynomial is said to have *degree* n, the maximum exponent of x.

Considering integers modulo 2^w, knowing how an integer can be factored by 2 is important, since any integer that can be factored by 2^w evaluates to 0, and any integer that can be factored by 2^i and is then multiplied by 2^j where $i + j \geq w$ likewise evaluates to 0. This is formalised in the definition below.

Definition 1. The *2-adic valuation* of an integer, is a function $\nu_2 : \mathbb{Z} \to \mathbb{N} \cup \{\infty\}$, such that for integer n, $\nu_2(n)$ is the largest r such that $n \equiv 0 \pmod{2^r}$, with $\nu_2(0) = \infty$. This then satisfies:

1. $\nu_2(ab) = \nu_2(a) + \nu_2(b)$,
2. $\nu_2(a + b) \geq \mathsf{min}(\nu_2(a), \nu_2(b))$.

For example, $\nu_2(8) = 3$, $\nu_2(12) = 2$, $\nu_2(6) = 1$ and $\nu_2(3) = 0$. The 2-adic valuation of an odd number is always 0. The following establishes a relationship between the bit-width and factorials, defining a value d_w whose factorial can be factored by 2^w.

Definition 2. For $w \in \mathbb{N}$, let d_w denote the smallest positive integer such that $\nu_2(d_w!) \geq w$.

For example, if $w = 2^k$ for some $k \geq 1$, $d_w = w + 2$, if $w = 2^3$, then $d_w = 10$.

2.1.1. Canonical and factorial bases

The usual representation of polynomials as a sum of monomials which are powers of x, as above, is referred to here as the *canonical basis* for polynomials. Polynomials do not have to be represented in this way.

Let:

$$\begin{aligned} x^{(0)} &= 1 \\ x^{(1)} &= x \\ x^{(i)} &= x(x-1)(x-2)...(x-i+1) \quad \text{if } i \geq 2 \end{aligned}$$

These terms are referred to as factorial monomials, and a polynomial can then be represented as a sum of these factorial monomials. This alternative representation is referred to as the *factorial basis*. The degree of a polynomial in the factorial basis is the degree of the factorial monomial with the greatest i and will denoted (i). It will be seen below that this is an attractive representation of polynomials.

2.2. Related work: binary polynomials

One of the dimensions of complexity discussed in the introduction was the degree of the polynomial. In [2] it was shown that for a given bit-width w, any polynomial can be represented with a maximum degree $d_w - 1$. Further, a polynomial given in the factorial basis is equivalent to one truncated to degree $(d_w - 1)$. Moreover, a polynomial given in the canonical basis can be converted to a representation in the factorial basis, truncated, then converted back, giving a canonical basis representation of maximum degree $d_w - 1$ in a normal form. These results are repeated and illustrated below along with an outline as to why they hold.

Key to the argument is the factorial basis. A factorial monomial represents a product in the form $x(x-1)(x-2)...(x-i+1)$, and when evaluated this becomes the product of a sequence of i consecutive numbers. The product of i consecutive numbers is divisible by $i!$. If $i \geq d_w$, then this product of consecutive numbers can be factored by 2^w, by the definition of d_w, hence evaluates to 0 modulo 2^w. That is, the result of evaluating $x^{(i)}$, with $i \geq d_w$, evaluates to 0. Hence any binary polynomial in factorial form whose degree is greater than or equal to d_w is equivalent to another whose degree is at most $d_w - 1$, indeed, in factorial form this is simply the truncation of higher degree terms. Since any polynomial over the factorial basis can be multiplied out to give a polynomial over the canonical basis, this maximum degree must also apply to binary polynomials over the canonical basis.

Given the bit-width it is possible to construct change of basis matrices that convert binary polynomials from the factorial basis to the canonical basis and vice versa. Taking the round journey of changing the basis from canonical to factorial and back again means that equivalent polynomials will be mapped to the same polynomial giving a normal form for binary polynomials.

To illustrate this, consider evaluating $3 + 3x + 2x^2 + 4x^3 + 2x^4 + 3x^6 + 2x^7$ for 4 bit numbers (that is, $w = 4$ and $d_w = 6$). The change of basis matrices for factorial to canonical (essentially multiplying out the factorial expression and reducing the coefficients mod 2^w) and canonical to factorial (the inverse of the first matrix) for degree 7 polynomials over w bits are given below.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 15 & 2 & 10 & 8 & 8 & 0 \\ 0 & 0 & 1 & 13 & 11 & 14 & 2 & 12 \\ 0 & 0 & 0 & 1 & 10 & 3 & 15 & 8 \\ 0 & 0 & 0 & 0 & 1 & 6 & 5 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 15 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 11 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 3 & 7 & 15 & 15 & 15 \\ 0 & 0 & 0 & 1 & 6 & 9 & 10 & 13 \\ 0 & 0 & 0 & 0 & 1 & 10 & 1 & 14 \\ 0 & 0 & 0 & 0 & 0 & 1 & 15 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying the second matrix to the input coefficient vector $(3, 3, 2, 4, 2, 0, 3, 2)$ and reducing modulo 2^4 gives $(3, 0, 7, 8, 1, 5, 13, 2)$. The i^{th} coefficient should be reduced modulo $2^{\mathsf{max}(w-\nu_2(i!),0)}$, following a similar argument to that above for maximum degree. Indeed, consider the sequence given by this expression for increasing i, starting at 0, which (for $w = 4$) starts $(2^4, 2^4, 2^3, 2^3, 2, 2, 1, 1)$. Hence here the maximum factorial degree of the polynomial is $(5) = (d_w - 1)$. Applying this 2-adic coefficient reduction gives $(3, 0, 7, 0, 1, 1, 0, 0)$. That is, an equivalent polynomial in the factorial basis over 4 bits is $3 + 7x^{(2)} + x^{(4)} + x^{(5)}$. Converting this back to the canonical basis using the first matrix gives a normal form polynomial of degree $5 \leq d_w - 1$ with coefficients (reduced modulo 2^4) $(3, 11, 0, 13, 7, 1, 0, 0)$, that is, $3 + 11x + 13x^3 + 7x^4 + x^5$. For this small example it is easy to verify that the polynomials are equivalent for 4 bits. Computationally, note that these matrices need only be calculated once and are then constants given bit-width w and maximum degree n. A full exposition and further examples can be found in [2], along with the generalisation to the multi-variate case.

2.3. Related work: polynomial evaluation

Polynomials are usually evaluated using Horner's rule [3, 4, 5] (Knuth notes that a similar method can be found in earlier work of Isaac Newton and of Qin Jiushao [6]). With multiplication being more expensive than addition, reducing the number of multiplications used in evaluating a polynomial pays off even at the cost of extra additions. The rule provides a straightforward rearrangement of a polynomial so that evaluation requires n multiplications and n additions.

$$a_0 + a_1x + a_2x^2 + ... + a_nx^n = a_0 + x(a_1 + x(a_2 + ... + a_nx)...)$$

For example:

$$3 + 3x + 2x^2 + 4x^3 = 3 + x(3 + x(2 + 4x))$$

Estrin's method [7, 5] recursively decomposes a polynomial and might be attractive for higher-order polynomials when some parallelism is available. When it is known that a polynomial is going to be repeatedly evaluated (for example, as part of an approximation of a basic trigonometric function) then further up front manipulation can give better performance, for example for integers modulo n polynomials of sufficiently large degree can be evaluated with $O(\frac{n}{2} + log(n))$ multiplications [8, 6].

2.4. Related work: bit-blasting polynomials

Bit-blasting, reducing a first-order SMT formula to a propositional SAT formula, remains a popular technique for handling the theory of bit-vectors. Each operator is encoded to its own circuit, either directly represented in CNF or using an intermediate form like And-Inverter Graphs (AIG) [9]. Then bit-level constant propagation and rewriting will be applied before the formula is passed to the SAT solver. In addition to the w propositional variables used to represent each first-order variable, intermediate variables are used extensively to reduce the number of clauses generated.

Many bit-vector operators bit-blast well. Operations that duplicate or move bits in a fixed patterns produce no propositional formula as they can be encoded using renaming. Other operations scale with the size of the bit-vectors, for example where w is the bit-width, `bvand` requires $3w$ clauses per bit and no auxiliary variables, whilst a `bvadd` requires $14w$ clauses and w auxiliary variables. Multiplication

behaves less well, with encodings of multiplication being $O(w^2)$. This is the major cost in polynomial evaluation and is a major concern.

Whilst there are numerous ways to encode multiplication, the straightforward approach is often used, implementing the shift-add algorithm:

```
bv multiplier_encoding(bv lhs, bv rhs) {
  int w = lhs.length();
  bv intermediate[w];

  intermediate[0] = and(repeat(w, lhs[0]), rhs);
  for (int i = 1; i < w; ++i) {
    intermediate[i] = add(intermediate[i-1],
                          lshift(and(repeat(w,lhs[i], rhs), i));
  }
  return intermediate[w-1];
}
```

The intermediate variables require $w \times w$ propositional variables, with $w \times (w-1)$ auxiliaries from additions. Then there are $w \times 3w$ clauses from bit-vector and, as well as $(w-1) \times 14w$ clauses from additions. If $w = 32$ then a multiplier under this quadratic scheme has 2016 variables and 16960 clauses. Reduction steps, such as using Wallace trees [10] (as done by Bitwuzla [9]), can reduce the number of full adders used. The long-standing Schönhage-Strassen conjecture [11] states that the lower bound for multiplication of w bit numbers is $w \log(w)$. This lower bound was reached in [12], however it is unclear whether this algorithm can be applied to bit-blasting in a way that is helpful.

Another property that is important for encodings is that they are propagation complete, that is, that all literals that are logically entailed can be inferred by unit propagation [13, 14]. Unfortunately the straightforward encoding of multiplication is not propagation complete and given that such an encoding could also be used for factoring it is unlikely that a polynomially sized propagation complete encoding of multiplication exists.

3. Polynomials modulo 2^n

This section makes some observations on polynomials over bit-vectors, noting that these are finite structures and can be decomposed in a tree-like way. Its main result is the construction of the polynomial over the high bits when the low bit is fixed. This will be exploited in the following section.

3.1. Polynomials over bit-vectors

First consider an example, the polynomial $f(x) = 3 + 3x + 2x^2 + 4x^3$ evaluated over four bit inputs, that is, modulo 2^4. This function is enumerated below ($x = [x_3, x_2, x_1, x_0]$):

x_3	x_2	x_1	x_0	$f(x)$			
0	0	0	0	0	0	1	1
0	0	0	1	1	1	0	0
0	0	1	0	0	0	0	1
0	0	1	1	1	0	1	0
0	1	0	0	1	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	1	0	1
0	1	1	1	0	1	1	0
1	0	0	0	1	0	1	1
1	0	0	1	0	1	0	0
1	0	1	0	1	0	0	1
1	0	1	1	0	0	1	0
1	1	0	0	0	1	1	1
1	1	0	1	0	0	0	0
1	1	1	0	0	1	0	1
1	1	1	1	1	1	1	0

Now suppose that instead of four bits $[x_3, x_2, x_1, x_0]$ the polynomial is to be evaluated over three bits, $[x_2, x_1, x_0]$. It is easy to see that the output of $f(x)$ is just the restriction of the output above to the low three bits (repeated twice). This can be seen below tabulated as a).

a)

x_2	x_1	x_0	$f(x)$		
0	0	0	0	1	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	1	0

b)

x_3	x_2	x_1	$f(x)$		
0	0	0	0	0	1
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	1	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	0	1	1
1	1	1	0	1	0

c)

x_3	x_2	x_1	$f(x)$		
0	0	0	1	1	0
0	0	1	1	0	1
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	0	1
1	1	0	0	0	0
1	1	1	1	1	1

What happens when instead of losing the top bit, the low bit is lost? The functions tabulated as b) and c) above give two slices of $f(x)$, in b) where the low input bit is 0, and in c) where the low bit is 1. These functions can be described as polynomials over 3 bits. That in b) is $1 + 3x + 4x^2$, and that in c) is $6 + 3x + 4x^2$. In general, is there a polynomial to describe the functions described by these slices, and if so what are they?

3.2. Subpolynomials

The slices above are referred to as *subpolynomials*. Binary polynomials that describe these can be constructed as demonstrated in the proposition below. The proposition shows how a binary polynomial over w bits leads to binary subpolynomials over the higher $w - 1$ bits. In the statement, the treatment has used a right-shift to describe the constant term. The subpolynomial of f resulting from fixing low bits $[b_j, ..., b_0]$ is a binary polynomial over $w - (j + 1)$ bits and will be denoted $f^{b_j \cdots b_0}$, for example, f^0, f^{110}, f^{011}.

Proposition 1. Suppose $b = [b_{w-1}, ..., b_1, b_0]$ is a bit-vector of bit-width w and that $f(x) = a_0 + a_1 x + ... + a_n x^n$ is a univariate polynomial over these w bits. For fixed low bit b_0 the subpolynomial over the $w - 1$ higher bits is denoted f^{b_0}. It is defined by slicing $f(x)$ on b_0 and is given by:

$$f^{b_0}(x) = \mathsf{shR}(f(b_0)) + \sum_{j=1}^{n} a_j f_j^s(x^j)$$

where each monomial maps as follows:

$$x^n \mapsto f_n^s(x^n) = 2^{n-1}x^n + b_0 \sum_{i=1}^{n-1} 2^{i-1} \binom{n}{i} x^i$$

In addition, the bit which is dropped by $\mathsf{shR}(f(b_0))$ is the output low bit of $f(b)$.

Proof. The input to the function is $b = [x_{w-1}, ..., x_1, b_0]$ (that is, the low bit is fixed). Consider $y = [x_{w-1}, ..., x_1]$, then $b = b_0 + 2y$. Consider $f(b) = f(b_0 + 2y) = a_0 + a_1(b_0 + 2y) + ... + a_n(b_0 + 2y)^n$. Expanding out each of the terms gives:

$$\begin{array}{l} \quad a_0 \\ +a_1 b_0 + 2a_1 y \\ +a_2 b_0^2 + 4a_2 b_0 y + 4a_2 y^2 \\ +a_3 b_0^3 + 6a_3 b_0^2 y + 6a_3 b_0 y^2 + 8a_3 y^3 \\ \quad \vdots \\ +a_j b_0^j + 2a_j \binom{j}{1} b_0^{j-1} y + ... + 2^i a_i \binom{j}{i} b_0^{j-i} y^i + ... + 2^j a_j \binom{j}{j} y^j \\ \quad \vdots \end{array}$$

Noting that the coefficients for terms involving y can always be factored by 2, the output low bit must be the low bit of the sum of the constant terms, that is, the remainder from the bit-shift. Bit-shifting the coefficients then gives the output for the higher bits, and renaming y to variable x leaves the results as given in the proposition. □

Notice in the proof of the proposition that if the input low bit is 0 then only the diagonal gives a non-zero output. Also note that since the subpolynomial is over $w - 1$ bits, the coefficients can be reduced modulo 2^{w-1}.

Applying this to the example polynomial $f(x) = 3 + 3x + 2x^2 + 4x^3$ when the input low bit is 0, the output $f^0(x)$ is below.

$$\begin{array}{rcl} 3 + 3x + 2x^2 + 4x^3 & \mapsto & \mathsf{shR}(3) + 3x + 2(2x^2) + 4(4x^3) \\ & = & 1 + 3x + 4x^2 + 16x^3 \\ & = & 1 + 3x + 4x^2 \end{array}$$

And when the low bit is 1, the output $f^1(x)$ is:

$$\begin{array}{rcl} 3 + 3x + 2x^2 + 4x^3 & \mapsto & \mathsf{shR}(3 + 3 + 2 + 4) + 3x + 2(2x + 2x^2) + 4(3x + 6x^2 + 4x^3) \\ & = & 6 + 3x + 4x + 4x^2 + 12x + 24x^2 + 16x^3 \\ & = & 6 + 19x + 28x^2 + 16x^3 \\ & = & 6 + 3x + 4x^2 \end{array}$$

This might be repeated giving, for example, $f^{00}(x) = 3x$ and $f^{11}(x) = 2 + 3x$.

It should be noted that the normalisation described in Section 2.2 has not been applied to the initial polynomial in this example.

3.3. Aside: permutation polynomials

Permutation polynomials over finite fields have been studied for a long time.

Definition 3. [15] A *permutation polynomial* is a polynomial over a finite field F_q of order $q = p^m$, $m \geq 1$ if and only the polynomials permutes the elements of F_q.

The standard definition is over finite fields and characterisation of what makes a polynomial a permutation polynomial has proved tricky. In contrast, [16] demonstrated that for binary polynomials (that is, where instead of finite field F_q, there is a finite ring of order 2^w) there is an elegant characterisation for when the polynomial is a permutation polynomial.

Proposition 2. [16] A polynomial is a permutation polynomial mod 2^w if and only if it has form $a_0 + a_1x + a_2x^2 + a_3x^3 + ...$ where $a_1 \pmod 2 = 1$, $a_2 + a_4 + ... \pmod 2 = 0$ and $a_3 + a_5 + ... \pmod 2 = 0$.

Observe that whilst most polynomials over bit-vectors are not permutation polynomials, an arbitrary polynomial is the sum of a permutation polynomial and a small, highly structured, polynomial.

Lemma 1. Given binary polynomial $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + ...$ over 2^w, then $f(x) = p(x) + q(x)$ where $p(x)$ is a permutation polynomial and $q(x) = ix + jx^2 + kx^3$, where $i, j, k \in \{0, 1\}$.

Proof. Consider the three conditions characterising a permutation polynomial from Proposition 2. Firstly, if $a_1 \pmod 2 = 1$, then $i = 0$, else $i = 1$. Secondly, if $a_2 + a_4 + ... \pmod 2 = 0$, then $j = 0$, else $j = 1$. Thirdly, if $a_3 + a_5 + ... \pmod 2 = 0$, then $k = 0$, else $k = 1$. Observe that following these rules ensures that $f(x) = p(x) + q(x)$, with $p(x)$ the permutation polynomial $p(x) = a_0 + (a_1 - i)x + (a_2 - j)x^2 + (a_3 - k)x^3 + a_4x^4 + ...$ □

4. Evaluating Binary Polynomials

The first step when evaluating a binary polynomial is to consider whether its degree can be reduced as in Section 2.2 and [2] and if so, reduce it. The rest of this section firstly details evaluating a binary polynomial for an input, and secondly covers how this might then be utilised within bit-blasting.

4.1. Evaluation of inputs

The method for deriving subpolynomials given in the previous section then suggests an approach to evaluating binary polynomials. For a given input $b = [b_{w-1}, ..., b_1, b_0]$, for values of i from 0 to $w - 1$ calculate the output i^{th} bit and derive the appropriate subpolynomial. Since the terms of the subpolynomials have constants that are rapidly rising powers of 2, whilst terms can be reduced mod 2^w with a decreasing w it can be seen that the order of subpolynomials drops rapidly during evaluation.

Returning to the worked example $f(x) = 3 + 3x + 2x^2 + 4x^3$ and supposing that the input is $[1, 0, 1, 1]$ then $f(1) = 12$ and the low output bit is 0, and as above, the subpolynomial to consider is $f^1(x) = 6 + 3x + 4x^2$.

The new input low bit is 1, so $f^1(1) = 13$, the output low bit is 1 and the next subpolynomial to consider is given by (reducing modulo 4):

$$\begin{aligned} 6 + 3x + 4x^2 &\mapsto \mathsf{shR}(13) + 3x + 4(2x^2 + 2x) \\ &= 6 + 11x + 8x^2 \\ &= 2 + 3x \end{aligned}$$

Next, the new input low bit is 0, so $f^{11}(0) = 2$, the output low bit is 0 and the next subpolynomial is (reducing modulo 2):

$$\begin{aligned} 2 + 3x &\mapsto \mathsf{shR}(2) + 3x \\ &= 1 + x \end{aligned}$$

Finally evaluate this at 1, giving $f^{011}(1) = 2$, giving output 0. That is, $f([1, 0, 1, 1]) = [0, 0, 1, 0]$, or in base 10, $f(11) = 2$ (noting that this output has wrapped).

As noted above, the bit-width drops by 1 with each subpolynomial, and the construction of the subpolynomials multiplies coefficients by powers of 2. This means that higher degree monomials

disappear from subpolynomials quite rapidly. If in the original there is a monomial x^n, noting that for each successive subpolynomial the x^n monomial is multiplied by 2^{n-1}, then in subpolynomial $f^{b_k \cdots b_0}$ there will not be a monomial in x^n when $k(n-1) > w - k$ since the 2-adic valuation coefficient of x^n in this subpolynomial will be greater than the bit-width of the bit-vector this subpolynomial is over. Rearranging the condition gives that $k > \frac{w}{n}$. Note that this is not independence of the subpolynomial from the higher degree monomial, since the constant is the result of the evaluation of the monomial for a previous subpolynomial.

Consider $3 + 3x + 2x^2 + 4x^3 + 2x^4 + 3x^6 + 2x^7$ evaluated over 32 bits. Then when $k > \frac{32}{7}$, for example, $k = 5$, the subpolynomial will have monomial $2^{1+(5\times6)}x^7$ over 27 bits, hence will always evaluate to 0. Again consider the monomial with x^2 and $k = 16$. This will have coefficient $2^{1+(16\times1)}$ over 16 bits, hence again will always evaluate to 0. That is, the higher 16 bits of the output bit-vector can be described by a linear subpolynomial, and this be exploited when bit-blasting.

4.2. Subpolynomial encoding for bit-blasting

The same principle can be used to produce a compact bit-level encoding of a binary polynomial. The least significant bit of the output can be produced using single bit `bvand` and `bvxor` to evaluate multiplication and addition respectively. Then the subpolynomial can be computed using the least significant bit of the input, allowing the whole encoding to be generated iteratively. It is important to note that the coefficients in the subpolynomial f^{b_i} will be symbolic expressions over the coefficient of the original binary polynomial f and each of the lower input bits $b_i, b_{i-1}, \ldots$. So simplification requires symbolic rewriting not pure calculations. One of the necessary rewrite rules is not implemented in all solvers. Using `lsb` to denote `(_ extract 0 0)` and `top` to denote `(_ extract w-1 1)` and `extend` to denote `(_ zero_extend w-2)` the rule is:

```
(top (bvadd x y)) = (bvadd (top x) (top y) (extend (bvand (lsb x) (lsb y)))
```

This allows the subpolynomials to simplify away some terms while still correctly modelling the chains of carries that can occur in rare circumstances.

An additional advantage of handling symbolic coefficients is that the encoding can be applied recursively to multivariate polynomials if they can be put into the appropriate nested form, such as $5(1+y) + x(3+4y+y^2) + x^2(2+3y^2)$. It remains an open question whether this is the most efficient approach to handling multivariate polynomials.

5. Experiments

The ultimate aim of any novel encoding is to improve either the number of problems decided or run-time in one or more class of problems, while minimising any negative effects on other classes. System-level evaluation of this sort can be challenging for a number of reasons – solver performance can be dependent on a large number of inter-related of factors, existing benchmark sets may not cover the class of problems improved by the encoding[1] and even if system-level improvements are shown, it is not always clear why they occur or how it connects to the changes made.

Reasoning about binary polynomials have all of these problem. As a first step to evaluating the technique described in Section 4 the following will be shown using synthetic benchmarks:

1. Reduction (Section 2.2) and the subpolynomial encoding (Section 4) reduce the size of the bit-blasted formulae.
2. They do not have an obvious negative effect on performance; simple problems remain simple.

[1]There is a social dynamic to this problem – if a class of problem is hard for current solvers, or thought to be hard, then it is likely that people will not "waste" time generating benchmarks for this class.

	Original/Native			Original/Subpolynomial			Reduced/Native			Reduced/Subpolynomial		
Degree	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses
2	0.00	126	377	0.00	83	242	0.00	126	377	0.00	83	242
3	0.00	344	1042	0.01	183	562	0.00	342	1036	0.01	183	562
4	0.00	610	1874	0.01	141	436	0.00	608	1861	0.01	142	441
5	0.01	786	2410	0.01	179	559	0.00	792	2425	0.00	179	559
6	0.00	978	3019	0.01	137	429	0.01	1003	3099	0.01	136	425
7	0.01	1231	3798	0.01	196	618	0.01	1254	3867	0.01	196	618
8	0.01	1397	4314	0.01	225	710	0.01	1346	4153	0.01	225	710
9	0.01	1663	5125	0.02	186	597	0.01	1415	4367	0.01	185	593
10	0.02	1654	5082	0.01	123	385	0.00	1169	3604	0.01	123	385
11	0.01	1952	6017	0.01	143	450	0.01	1384	4269	0.02	141	444
12	0.02	2130	6574	0.01	202	636	0.01	1537	4739	0.00	207	647
13	0.01	2180	6729	0.01	187	586	0.01	1237	3816	0.01	178	560
14	0.01	2386	7346	0.02	188	595	0.00	1327	4093	0.01	188	595
15	0.01	2582	7985	0.01	177	565	0.00	1198	3693	0.02	174	553
16	0.03	2639	8169	0.02	182	570	0.01	1317	4062	0.01	155	485

Table 1
Instance sizes and run-times for Bitwuzla 0.7.0 with 8-bit polynomials on root finding benchmarks

5.1. Benchmarks

A benchmark generator has been written using the cvc5 API. It generates permutation polynomials of a given degree that have coefficients randomly chosen from bit-vectors of given width. These are used to create four SMT-LIB files with different encodings of the polynomial:

1. The original polynomial in canonical form using `bvmul` and `bvadd` so that the solver's native simplification and encoding is used.
2. The original polynomial represented using the subpolynomial encoding.
3. The reduced polynomial expressed in canonical form so the solver's native simplification and encoding is used.
4. The reduced polynomial represented using the subpolynomial encoding.

In each of these a single assertion requires that the output must be zero (i.e. finding a root of the polynomial). Permutation polynomials have exactly one root. Experiments with other families of polynomials suggest that there are classes of polynomial for which root finding is easier (in some case solvable with simplification alone). So permutation polynomials represent a "hardest case" for these techniques. [2]

5.2. Results

Bitwuzla [9] 0.7.0 is used as the leading bit-vector solver. Again this means any improvements are a lower bound, other solvers may have larger improvement. Evaluation was done using a 1.70GHz Intel Core i5-4210U, running Debian GNU/Linux 11. CPU time was limited to 60 seconds and memory limited to 1GB, although neither of these affected the results. Tables 1, 2 and 3 show the run time (in seconds) and the number of Boolean variables and clauses after bit-blasting for 8 bit, 16 bit and 32 bit polynomials over a range of degrees.

By comparing the original and reduced polynomials using the native encoding, it is possible to see that the reduction makes a significant difference to the variables and clauses used. The size of the original polynomial continues to rise as the degree rises. However the reduced polynomial encoding sizes stop rising when the bit-width reaches $w + 2$, as predicted by the theoretical results in [2].

Comparing the native and subpolynomial encodings there is a significant reduction in the size of the encoding for all except the first few degrees in the larger bit-widths. In the most extreme cases

[2]The generator and benchmarks are available at https://github.com/martin-cs/subpolynomial-encoding

	Original/Native			Original/Subpolynomial			Reduced/Native			Reduced/Subpolynomial		
Degree	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses
2	0.00	965	2905	0.02	1105	3344	0.01	965	2905	0.02	1105	3344
3	0.01	2149	6468	0.03	1979	6073	0.01	2149	6468	0.03	1979	6073
4	0.02	2841	8563	0.03	1989	6124	0.01	2837	8550	0.03	1989	6124
5	0.02	4828	14658	0.06	2592	8009	0.02	4826	14651	0.04	2592	8009
6	0.03	5926	17972	0.07	2623	8105	0.04	5919	17946	0.08	2567	7930
7	0.04	7100	21554	0.06	2933	9100	0.03	7087	21509	0.08	2765	8581
8	0.04	8392	25487	0.07	2702	8382	0.03	8313	25244	0.07	2705	8393
9	0.04	8715	26421	0.07	2604	8089	0.04	8781	26625	0.06	2589	8048
10	0.06	10460	31723	0.08	2644	8222	0.04	10342	31361	0.09	2643	8218
11	0.07	11931	36246	0.08	2916	9060	0.05	11751	35693	0.09	2916	9060
12	0.05	12531	38057	0.08	2935	9119	0.05	12404	37680	0.08	2934	9115
13	0.05	13358	40554	0.11	2793	8690	0.06	13297	40364	0.09	2793	8690
14	0.06	14434	43798	0.10	2614	8138	0.06	14475	43922	0.10	2615	8142
15	0.09	16297	49522	0.09	2665	8293	0.10	16378	49756	0.09	2667	8301
16	0.10	16895	51373	0.10	2581	8042	0.07	16534	50249	0.10	2581	8039
17	0.07	18369	55826	0.12	2655	8284	0.07	18100	54994	0.11	2659	8286
18	0.09	19328	58714	0.11	2740	8521	0.07	17904	54401	0.10	2805	8720
19	0.09	20371	61857	0.13	2651	8260	0.07	17803	54094	0.13	2681	8340
20	0.13	21437	65074	0.09	2555	7969	0.07	17799	54084	0.11	2700	8411
21	0.08	22447	68125	0.11	2748	8560	0.09	17806	54103	0.12	2755	8582
22	0.10	23640	71722	0.11	2754	8561	0.07	17601	53481	0.11	2773	8621
23	0.19	25210	76602	0.13	2884	8969	0.09	15302	46502	0.13	2884	8969
24	0.10	26157	79508	0.11	2732	8507	0.07	15872	48227	0.13	2719	8465
25	0.12	27704	84170	0.12	2670	8306	0.07	16881	51285	0.09	2650	8243
26	0.15	28820	87587	0.11	2775	8623	0.07	18014	54740	0.16	2802	8702
27	0.20	29005	88173	0.10	2761	8578	0.10	16715	50795	0.11	2691	8368
28	0.14	30336	92225	0.11	2745	8539	0.07	14712	44716	0.11	2744	8534
29	0.16	32211	97922	0.11	2813	8748	0.07	15972	48547	0.11	2901	9009
30	0.25	32896	99971	0.09	2988	9297	0.13	18566	56413	0.12	2976	9261
31	0.14	32647	99139	0.10	2632	8187	0.09	18665	56712	0.12	2637	8200
32	0.25	34818	105802	0.12	2302	7165	0.15	17988	54659	0.13	2274	7080

Table 2
Instance sizes and run-times for Bitwuzla 0.7.0 with 16-bit polynomials on root finding benchmarks

the subpolynomial encodings are 20 times smaller. However typical cases are 3 to 10 times smaller. Critically this is achieved without significant increase in solving time.

Finally comparing the original and reduced polynomials using the subpolynomial encodings shows a very minimal benefit. This is likely because the higher degree terms that would definitely be removed during reduction would also be removed during the first few subpolynomial computation steps. However, notice the size of the subpolynomial encoding in terms of both variables and clauses is almost independent of the degree, reflecting that higher degree terms quickly drop out of subpolynomials.

6. Discussion and Conclusion

This paper recaps previous results about the effective degree limit and reduction of binary (bit-vector) polynomials as well as proposing a new means of evaluation. Bit-blasting representations of binary polynomials using this evaluation method have been shown to be significantly more compact than Bitwuzla's native encoding and not obviously detrimental to performance in synthetic benchmarks. Significant future work is needed to show that these encodings can produce system-level performance improvements. It will be necessary to extend subpolynomial evaluation to handle multivariate polynomials, to integrate this approach within a solver's bit-blasting engine (including identifying / constructing polynomials from the solver input) and to identify suitable sets of "real world" benchmarks where

	Original/Native			Original/Subpolynomial			Reduced/Native			Reduced/Subpolynomial		
Degree	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses
2	0.01	5721	17203	0.07	8546	25914	0.02	5721	17203	0.11	8546	25914
3	0.04	11304	33987	0.14	16304	49540	0.04	11303	33983	0.19	16304	49540
4	0.07	17317	52222	0.24	20864	63428	0.07	17317	52224	0.20	20846	63352
5	0.14	22958	69251	0.29	25110	76384	0.14	22960	69259	0.24	25106	76371
6	0.10	28452	85780	0.27	27268	83009	0.11	28420	85686	0.26	26957	82058
7	0.31	32928	99300	0.30	27121	82653	0.21	32978	99446	0.38	27297	83180
8	0.16	39848	120176	0.29	29088	88606	0.21	39837	120137	0.29	28980	88270
9	0.36	45177	136320	0.45	30324	92557	0.32	45077	136005	0.40	30291	92448
10	0.38	51271	154704	0.42	31222	95275	0.37	51267	154686	0.39	31274	95433
11	0.20	56039	169098	0.52	32191	98226	0.34	56076	169210	0.45	32135	98045
12	0.32	58593	176603	0.48	31913	97518	0.28	58422	176069	0.48	31928	97560
13	0.43	66933	201931	0.57	32548	99459	0.49	66457	200471	0.72	32703	99929
14	0.31	71685	216269	0.56	32167	98281	0.43	71751	216461	0.54	32148	98220
15	0.32	75813	228757	0.59	31172	95282	0.33	76081	229558	0.54	31158	95231
16	0.66	85013	256559	0.62	32499	99356	0.89	84454	254861	0.55	32413	99098
17	0.46	85331	257425	0.66	31931	97616	0.52	85097	256722	0.58	31930	97614
18	0.64	95407	287938	0.61	32756	100089	0.67	95064	286887	0.61	32761	100103
19	0.57	100899	304504	0.75	32600	99672	0.65	100294	302660	0.70	32579	99606
20	0.74	104962	316657	0.79	31620	96713	1.12	104789	316140	0.79	31646	96789
21	0.76	112976	340954	0.87	32274	98670	0.84	112155	338451	1.00	32264	98639
22	0.91	117728	355309	0.89	32633	99773	0.87	116816	352534	0.85	32629	99759
23	0.52	120023	362269	1.08	32793	100224	0.51	118972	359061	0.96	32793	100222
24	1.24	129115	389675	1.00	32735	100047	0.92	128533	387898	1.03	32740	100057
25	1.27	130632	394033	0.96	29632	90662	0.93	130017	392166	0.98	29618	90626
26	0.88	136169	410879	1.08	30708	93890	1.14	133574	403030	1.04	30716	93918
27	1.26	141941	428387	0.95	30928	94568	1.20	141833	428047	1.08	30906	94500
28	1.31	148429	447993	1.09	32499	99335	1.16	146564	442348	1.09	32549	99477
29	0.74	154484	466274	1.16	32163	98345	0.71	154211	465393	1.14	32271	98684
30	1.49	160911	485668	1.13	32667	99806	1.21	158987	479834	1.19	32641	99728
31	1.19	163806	494425	1.15	31015	94847	1.12	161317	486918	1.15	30973	94724
32	1.53	170248	513862	1.09	32597	99610	1.43	168765	509335	1.08	32576	99539
33	1.08	174777	527420	1.10	30229	92487	1.07	162950	491788	1.19	30234	92499
34	0.84	176098	531555	1.19	31836	97329	0.99	169348	511127	1.15	31754	97077
35	0.89	186531	562889	1.32	33080	101035	0.79	171454	517485	1.15	33182	101352
36	1.52	190926	576302	1.16	30355	92854	0.99	163489	493398	1.13	30476	93212
37	1.04	200838	606176	1.13	32346	98893	0.76	162516	490504	1.14	32381	99008
38	0.99	202910	612455	1.11	32650	99816	0.93	176397	532361	1.12	32560	99554
40	1.85	213865	645519	1.41	31236	95405	1.33	171426	517367	1.16	31146	95132
42	1.10	225025	679214	1.13	31980	97810	0.86	174218	525791	1.16	31893	97548
44	1.53	237393	716519	1.14	31809	97235	1.06	166587	502753	1.15	31751	97078
46	2.48	248453	749918	1.06	32278	98655	1.04	173588	523909	1.05	32410	99045
48	1.42	256100	772718	0.98	31153	95141	0.77	163285	492745	1.04	31533	96312
50	2.70	266756	805180	1.14	32403	99078	1.04	163517	493514	1.09	32350	98897
52	2.81	279725	844277	1.23	31590	96576	1.32	174207	525768	1.16	31529	96389
54	1.73	289798	874732	1.11	29425	90005	1.13	172798	521529	1.14	29430	90013
56	2.11	299986	905250	1.18	31407	96021	1.08	171508	517639	1.12	31000	94773
58	3.17	312945	944579	1.55	33356	101949	1.13	173888	524794	1.10	33286	101729
60	1.80	325833	983475	1.18	31626	96714	0.83	174646	527075	1.09	31669	96860
62	1.91	337210	1017841	1.16	31769	97134	0.84	174588	526920	1.15	32615	99712
64	2.38	339605	1024929	1.20	32448	99223	1.24	173881	524762	1.19	32506	99385

Table 3
Instance sizes and run-times for Bitwuzla 0.7.0 with 32-bit polynomials on root finding benchmarks

reasoning about binary polynomials is central to the problem difficulty. This is first bit-blasting encoding of this kind and so further improvements are likely possible.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. Brain, Further Steps Down The Wrong Path: Improving the Bit-Blasting of Multiplication, in: Proceedings of the 19th International Workshop on Satisfiability Modulo Theories, volume 2908 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 23–31.

[2] A. Gàmez-Montolio, E. Florit, M. Brain, J. M. Howe, Efficient Normalized Reduction and Generation of Equivalent Multivariate Binary Polynomials, in: Workshop on Binary Analysis Research (BAR) 2024, NDSS, 2024, pp. 1–12.

[3] W. G. Horner, A new method of solving numerical equations of all orders, by continuous approximation, Philosophical Transactions of The Royal Society 109 (1819) 308–335.

[4] F. de Dinechin, M. Kumm, Application-Specific Arithmetic, Springer, Cham, 2024.

[5] J.-M. Muller, Elementary Functions: Algorithms and Implementation (2nd ed.), Birkhäuser, Boston, 2006.

[6] D. E. Knuth, The Art of Computer Science, volume 2 (3rd ed), Addison Wesley, 1998.

[7] G. Estrin, Organization of Computer Systems: the Fixed Plus Variable Structure Computer, in: Western Joint IRE-AIEE-ACM Computer Conference, Association for Computing Machinery, New York, 1960, p. 33–40.

[8] M. O. Rabin, S. Winograd, Fast Evaluation of Polynomials by Rational Preparation, Communications on Pure and Applied Mathematics XXV (1972) 433–458.

[9] A. Niemetz, M. Preiner, Bitwuzla, in: Computer Aided Verification, volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17.

[10] C. S. Wallace, A suggestion for a fast multiplier, IEEE Transactions on Electronic Computers EC-13 (1964) 14–17.

[11] A. Schönhage, V. Strassen, Schnelle Multiplikation großer Zahlen, Computing 7 (1971) 281–292.

[12] D. Harvey, J. van der Hoeven, Integer multiplication in time O(n log n), Annals of Mathematics 193 (2021) 563–617.

[13] L. Bordeaux, J. Marques-Silva, Knowledge Compilation with Empowerment, in: SOFSEM 2012: Theory and Practice of Computer Science, volume 7147 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 612–624.

[14] M. Brain, L. Hadarean, D. Kroening, R. Martins, Automatic Generation of Propagation Complete SAT Encodings, in: Verification, Model Checking, and Abstract Interpretation, volume 9583 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 536–556.

[15] R. Lidl, G. L. Mullen, When Does a Polynomial over a Finite Field Permute the Elements of the Field?, American Mathematical Monthly 95 (1988) 243–246.

[16] R. L. Rivest, Permutation Polynomials Modulo 2^w, Finite Fields and Their Applications 7 (2001) 287–292.